TD PG106
gdb

Ce TD est consacré à la découverte de l'outil gdb.

1 Gdb : premiers pas

Dans le programme `gdb` (une fois la commande `gdb` lancée dans le terminal), la commande `help` permet de consulter la documentation des commandes internes à `gdb`.

L'outil `gdb` est un outil très complet (la documentation fait plus de 900 pages !).

Il est fortement recommandé de garder sous la coude l'aide mémoire disponible ici : `https://sourceware.org/gdb/download/onlinedocs/refcard.pdf`

▸**Exercice 1.** *Préparation :*

Pour que l'on puisse travailler efficacement avec `gdb`*, il faut que notre code soit compilé avec des informations additionnelles (symboles de debug). C'est le compilateur qui introduit ces symboles lors de la phase de compilation. L'option* `-g` *permet de demander à* `gcc` *l'ajout de ces symboles. Il est également recommandé de désactiver les optimisations de code : c'est l'option* `-O0` *(lettre o majuscule suivie du chiffre zéro) qui permet cela*(1)*.*

Modifier votre `Makefile` *pour que les options indiquée ci-dessus soient bien positionnées.*

Recompiler votre code (`make clean; make all`*) et charger l'exécutable* `prog` *(produit à partir des fichiers* `.o`*) dans* `gdb` *:* `gdb prog`*. Vérifier que* `gdb` *a bien trouvé les symboles de debug.*

▸**Exercice 2.** *Exécution et affichage :*

La première commande à connaître dans `gdb` *est la commande* `run`*. Celle-ci permet dans lancer l'exécution du programme chargé (elle peut être suivie d'arguments qui seront transmis au programme). Tester cette commande.*

La deuxième commande est `list`*. Celle-ci permet de demander d'afficher des portions de code source (grace aux symboles de debug,* `gdb` *peut associé le code machine au code source à l'origine de celui-ci).*

`list` *peut être suivie de plusieurs arguments dont :*

— *le numero d'une ligne (par rapport au fichier courant*(2)

— *le nom d'un fichier source avec optionnellement un numéro de ligne*

— *le nom d'une fonction*

Consulter la documentation de cette commande et utiliser celle-ci pour :

— *afficher le code principal du programme*

— *afficher le code du fichier* `list.c`

— *afficher le code de la fonction* `hashAdd`

Comment faire pour que `list` *affiche plus de 10 lignes à chaque appel ?*

Pour finir cet exercice, utiliser la commande `tui enable` *(la commande* `refresh` *permet de mettre à jour l'affichage si celui-ci est* buggé*,* `Ctrlx o+` *permet de basculer du source au prompt).*

——

Il n'y a que deux états dans `gdb` : soit le programme s'exécute et l'on n'a pas la main dans gdb, soit le programme est arrêté et dans ce cas on peut analyser l'état du programme.

——

(1). à l'opposée `-O3` va permettre de produire un code plus optimisé en performance

(2). le *fichier courant* correspond au fichier source associé au code en cours d'exécution. Sans exécution en cours, c'est le fichier contenant la fonction `main`

▸**Exercice 3.** *Breakpoints et exécution pas à pas*

Un breakpoint *est un point dans le code auquel on souhaite que l'exécution du programme soit suspendu.*

Avec la commande `break` *positionner un point d'arrêt sur la fonction* `main` *: donner deux façons de faire cela (*`help break`*).*

Exécuter (`run`*) à nouveau votre programme : celui-ci va s'arrêter lorsqu'il va rencontrer le point d'arrêt. À partir de là, entrer la commande* `continue` *: le programme reprend son exécution jusqu'au prochain point d'arrêt. Relancer le programme.*

Pour avancer d'une ligne (exécution des instructions associées à la ligne de code courante), utilisez la commande `next`*.*

Si une ligne comporte un appel de fonction, la commande `next` *ne vous fera pas renter dans cette fonction, la commande* `step` *si.*

Procéder à un plusieurs avancements avec `next` *jusqu'à atteindre l'appel à la fonction* `hashAction` *(si vous l'avez dépassé, refaire* `run` *puis des* `next`*).*

A cette ligne, utiliser la commande `step` *: cela nous permet de faire un pas en entrant dans la fonction. Vous pouvez continuer l'exécution pas à pas avec* `next`*. Pour continuer jusqu'à la sortie de la fonction courante utiliser la commande* `finish`*.*

Vous devrier vous trouver à nouveau dans le `main`*, juste après l'appel à* `hashAction`*. Essayer les commandes* `step` *(on entre dans* `teachingInit`*),* `finish`*,* `step` *(on entre dans* `hashAdd`*),* `finish`*, puis* `step` *à nouveau.*

Nous nous trouvons à vouloir suivre l'exécuction de la fonction `printf`*. Le problème est que nous n'avons pas les fichiers sources qui implémentent cette fonction : on se retrouve* aveugle*. Vous pouvez continuer à l'aveugle avec* `step` *ou* `next` *ou bien sortir avec* `finish`*.*

Pour pousuivre l'exécution jusqu'à une ligne précise (et sans positionner de breakpoint), vous pouvez utiliser la commande `until`*.*

Pour lancer le programme et vous arrêter au début du `main`*, vous pouvez utiliser* `start` *(fonctionne comme run mais s'arrête à la première instruction).*

Enfin, vous pouvons mettre fin à l'exécution en cours avec la commande `kill`*.*

——

Nous avons vu les commandes suivantes :

— run : démarrer une exécution
— break : placer un point d'arrêt dans le code
— next : exécuter la ligne courante afin d'atteindre la ligne suivante
— step : exécuter la ligne courante en entrant dans les appels de fonction
— finish : continuer jusqu'à la sortie de la fonction courante
— continue : poursuivre l'exécution jusqu'au prochain point d'arrêt
— until : avance jusqu'à une ligne
— start : démarre le programme puis l'arrêt à la première instruction

——

▸**Exercice 4.** *Affichage*

Pour afficher la valeur d'une variable, on dispose de deux commandes : `print` *ou bien* `display`*. Quelle est la différence entre ces deux commandes (cf* `help`*) ?*

Placer un point d'arrêt sur la fonction `hashAdd` *et relancer une exécution du programme.*

Afficher la valeur de l'argument `hash` *à l'aide de print. Vous pouvez afficher le contenu mémoire à l'adresse contenue dans* `hash` *:* `print *hash`*.*

Essayer d'afficher le contenu de `hash->entries` *:* `gdb` *ne sait pas que c'est un tableau (ce pourrait être un pointeur vers uniquement une seule cellule). Tester la commande* `print (*hash->entries)@13` *ou encore mieux* `print (*hash->entries)@hash->len`*.*

Concernant le deuxième argument `data`*, afficher la valeur de celui-ci. Essayer d'afficher le contenu mémoire à l'adresse contenue dans* `data` *(*`print *data`*). On peut forcer l'interprétation en utilisant un*

`cast` : `print *(struct Person *)data`.

Passons à la variable locale `value` *: afficher la valeur de celle-ci. Nous allons avancer pas à pas dans la fonction et l'on souhaite pouvoir surveiller* `value`*, entrer la commande* `display value`*. Vous pouvez avancer dans l'exécution avec plusieurs appels à* `next` *et observer le changement de la valeur de la variable.*

Pour terminer, sachez que l'on peut faire un appel de fonction avec ces deux fonctions (`print` *et* `display`*). Essayer les commandes suivantes :* `print personPrint(data)` *ou encore* `print printf("hello!")`*. Ceci doit être utiliser avec précaution car les appels peuvent modifier l'état mémoire de votre processus et donc avoir des effets sur la suite de l'exécution.*

——

Nous avons vu les commandes suivantes :

— print
— display

——

▸**Exercice 5.** *Pile et fenêtre !*

Placer un breakpoint sur la fonction `cellAlloc` *et exécuter le programme puis utiliser la fonction* `continue` *jusqu'à atteindre un appel à* `cellAlloc`*.*

Utiliser la commande `backtrace` *: cela permet d'afficher la pile d'appels de fonctions. Chaque ligne correspond à l'appel d'une fonction, on appelle cela une* frame *de la pile.*

Vous pouvez consulter les variables locales de la frame *courante (au moment de l'arrêt, c'est la dernière fonction appelée).*

Pour également changer de frame*, en utilisant les commandes* `up` *(pour remonter dans la fonction appelante) et* `down` *(pour redescendre). Noter bien que l'exécution du programme est toujours suspendu, c'est juste le point d'analyse courant dans la pile qui change.*

Utiliser ces commandes pour connaître la valeur de la variable locale `value` *de la fonction* `hashAdd`*. Afficher également la valeur de la variables* `teachings` *du* `main` *et finir par afficher la valeur de la variable locale* `c` *de* `cellAlloc`*.*

——

Nous avons vu les commandes suivantes :

— backtrace
— up
— down

——

▸**Exercice 6.** *Arrêt conditionnel*

Consulter la documentation de la commande `cond`*.*

Placer un point d'arrêt sur la fonction `teachingInit` *et faire en sorte que l'on s'arrêt à l'appel de cette fonction uniquement si* `code` *et supérieur à* 150*.*

Les conditions sont très pratiques lorsque le code passe de nombreuse fois par un endroit que l'on souhaite analyser mais que seulement un (ensemble de) cas nous intéresse.

La commande `info` *permet de connaître l'état sur les fonctions internes à* `gdb`*. Tester* `info breakpoints` *: nous pouvons voir la liste des points d'arrêt, savoir s'ils sont actifs et s'ils sont conditionnés.*

Placer un point d'arrêt sur `cellAlloc` *puis relancer le programme : il va s'arrêter à chaque passage par la fonction. Si l'on souhaite désactiver le point d'arrêt, on utilise* `disable 1` *(si* 1 *est le numéro du point d'arrêt). Vérifier le status du point d'arrêt avec* `info`*. On pourra le réactiver avec* `enable 1` *plus tard. Enfin, on peut le supprimer définitivement avec* `delete 1`*.*

——

Nous avons vu les commandes suivantes :

— cond
— info breakpoints

— enable
— disable
— delete

——

▸**Exercice 7.** *Surveillance*

Consulter la documentation de la commande `watch`.

Voici un cas d'usage : repartir de zéro (quitter puis relancer `gdb`*), placer un point d'arrêt sur le* `main` *et lancer le programme. Passer la première ligne et afficher* `*(people->entries)@13`.

On souhaite continuer l'exécution jusqu'à l'appel à `hashAction`. *Lister le source* `list` *et identifier la ligne correspondante à cette appel (disons que c'est 94), vous pouvez poursuivre l'exécution jusqu'à cette ligne sans placer de breakpoint en utilisant* `until 94`.

Afficher à nouveau `*(people->entries)@13` *: on voit que* `people->entries[7]` *n'est plus à* `NULL`. *On va identifier précisement le moment auquel cette case à changer de valeur grace à* `watch`.

Reprendre l'exécution du programme depuis le début. Passer juste la première ligne (initialisation de `people`*) puis afficher* `people->entries[7]` *: celle-ci est bien à* `NULL`. *Entrer la commande* `watch -l people->entries[7]` *puis pousuiver l'exécution (*`continue`*). L'exécution va s'arrêter* **juste après** *la ligne qui a déclenché le changement de valeur : on peut alors utiliser toutes les commandes vu ci-dessus pour inspecter la situation (*`backtrace,up,down,print`*...).*

La commande `watch -l` *est très puissante pour identifier rapidement l'origine d'une* incohérence *mémoire : une variable a une valeur qui n'est pas celle que vous attendez.*

——

Nous avons vu les principales commandes à connaître dans `gdb`. Sachez que chaque commande peut être utilisée en version abbrégée : par exemple `c` pour `continue`, `bt` pour `backtrace` .... La documentation de chaque commande indique sur la première ligne les versions abbrégées disponibles pour la commande.

——

▸**Exercice 8.** *Un premier bug dans* `hash`

Dans `prog.c` *ajouter le code suivant :*

```
printf("ajout␣%d␣\n",hashAdd(people,personInit("Johnny","L01.")));

struct Person *p=personInit("Alexandre","Mur");
printf("search␣%d␣\n",hashSearch(people,p));

personFree(p);

hashAction(people,(void *)personPrint);
```

La fonction `hashSearch` *est censée renvoyer* 0 *si l'élément est présent dans la table et* 1 *si ce n'est pas le cas. Ici, l'appel renvoie* 0 *alors qu'Alexandre Mur n'a pas été ajouté à la table !*

Utiliser `gdb` *pour comprendre et corriger ce bug.*

2 Git : les branches orphelines

Il est possible de créer des branches qui n'ont pas de point d'histoire commun : un peu comme un dépôt nouvellement crée. On appelle cela des branches orpheline.

▸**Exercice 9.** *Création d'une branche orpheline*

Assurer vous que votre travail en cours est bien `commit` *(et* `push` *tant qu'à faire). La commande* `git status` *doit vous indiquer* `nothing to commit, working tree clean`.

Nous allons créer une branche sans que celle-ci soit connectée à la branche courante : `git checkout --orphan exo_gdb` *: cette commande créer la branche orpheline et nous place directement dedans.*

Si vous faites un `git status` *vous verrez des fichiers en attente de commit, nous allons* purger *cela avec la commande* `git rm --cached -r .`*, la commande* `git status` *vous indique alors les fichiers comme ne faisant pas parti du dépôt. On peut maintenant effacer les fichiers* `rm -rf *` *et éventuellement ajouter le fichier* `.gitignore` *à la branche (*`add` *suivi d'une* `commit`*).*

Nous disposons maintenant d'une branche orpheline : nous allons faire les exercices suivants dedans.

3 Gdb : training

▸**Exercice 10.** *Entrainement !*

Entrainez-vous sur les autres fichiers sources (ils sont dans une branche de votre dépôt : à vous de les trouver) dans l'ordre suivant :

- `01space.c`
- `02xtrem.c`
- `03foo.c`
- `04alloc.c`
- `05lists.c`
- `06hack.c` *(bugs assez complexes !).*
- `07geek.c` *(bugs assez complexes !).*

Vous pouvez vous entrainer en dehors des séances sur les autres programmes (bugs plus complexes).